The Life Changing HERMIT: A Case Study of the Worker/Wrapper Transformation

By

Brad Torrence

Submitted to the Department of Electrical Engineering and Computer Science and the
Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering

Dr. Andy Gill, Chairperson

Committee members

Dr. Alexander Perry

Dr. Prasad Kulkarni

Date defended: 30 January 2015

The Thesis Committee for Brad Torrence certifies
that this is the approved version of the following thesis :

The Life Changing HERMIT: A Case Study of the Worker/Wrapper Transformation

Dr. Andy Gill, Chairperson

Date approved: 30 January 2015

Abstract

In software engineering, altering a program's original implementation disconnects it from the model that produced it. Reconnecting the model and new implementations must be done in a way that does not decrease confidence in the design's correctness and performance. This thesis demonstrates that it is possible, in practice, to connect the model of Conway's Game of Life with new implementations, using the worker/wrapper transformation theory. This connection allows development to continue without the sacrifice of re-implementation.

HERMIT is a tool that allows programs implemented in Haskell to be transformed during the compilation process, and has features capable of performing worker/wrapper transformations. Specifically in these experiments, HERMIT is used to apply syntax transformations to replace Life's linked-list based implementation with one that uses other data structures in an effort to explore alternative implementations and improve overall performance.

Previous work has successfully performed the worker/wrapper conversion on an individual function using HERMIT. This thesis presents the first time that a programmer-directed worker/wrapper transformation has been attempted on an entire program. From this experiment, substantial observations have been made. These observations have led to proposed improvements to the HERMIT system, as well as the formal approach to the worker/wrapper transformation process.

Acknowledgements

I would like the thank Andrew Farmer for his help understanding HERMIT. And a special thanks to Professor Andy Gill for his help understanding the worker/wrapper transformation theory, as well as the opportunity to explore this material.

This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

Contents

Chapter 1

Introduction

The worker/wrapper transformation theory, introduced by Gill & Hutton (2009), is the driving force behind the experiments demonstrated in this thesis. Transformation capabilities based on the worker/wrapper methodology have been implemented in the Haskell Equational Reasoning Model to Implementation Tunnel (HERMIT). It has already demonstrated several examples of these abilities (Farmer et al., 2012; Sculthorpe et al., 2013; Farmer et al., 2014). However, those examples consisted of small transformations over a single function. In this thesis, HERMIT's capabilities, using worker/wrapper methods, are scaled to support a larger example – an implementation of the Game of Life.

Prior to the experiments in program transformation began, a suitable program had to be selected. The Game of Life was chosen because it has a simple definition and is uniform in its use of a primary data structure. The uniform use of a single data structure provides the perfect target for the transformation.

A basic formula to correctly execute worker/wrapper transformations exists (Gill & Hutton, 2009), and this formula is followed to correctly perform the transformations using the HERMIT syntax rewriting tool. HERMIT has been designed with capabilities that support safe worker/wrapper transformations. This thesis will demonstrate how the worker/wrapper formula is currently applied by HERMIT, to transform an individual function. The thesis will also demonstrate that

(with a few extra considerations) this process can be repeated for all of the functions in a module to correctly transform it.

The process was successfully completed for three different program transformations. The results of these transformations show that improvements can be made through this transformation process. However, to successfully repeat this process, the recipe for function transformations must be updated. The original recipe is modified in this thesis to consider multi-function transformations. This is in addition to a proposed recipe for the correct transformation of multi-function modules and multi-module programs.

Chapter 2

Background

2.1 The Worker/Wrapper Transformation

The worker/wrapper transformation is a technique for changing the implementation of a function. The transformation results in two function definitions: the worker and the wrapper. The worker function performs the new implementation of the original computation. And the wrapper connects the new implementation with the original interface by calling the worker and transforming its result.

The worker/wrapper transformation requires some basic assumptions to support the process (Gill & Hutton, 2009). The first assumption requires the existence of two key functions, `rep` and `abs`, which translate the original representation `A` to/from the new representation `B`, respectively. Defining these functions so that one reverses the operation of the other, leads to the assumption:

```
rep . abs = id
```

Although in practice, this assumption may not hold unless applied to a particular context, in which case, a weaker assumption may suffice using a given function, `f`:

```
fix (rep . abs . f) = fix (id . f)
```

Given the preconditions on the relationship between `abs` and `rep`, we can assume another condition that correctly performs the worker/wrapper transformation (Sculthorpe & Hutton, 2014).

Specifically, given the original computation `body`, the transformation functions `rep` and `abs`, and a new implementation of the original computation `work`, where:

```
work = rep . body . abs
```

Then using the fixed-point calculation, the following condition can safely be assumed:

```
fix work = fix (rep . body . abs)
```

And the original computation can be rewritten as worker and wrapper functions:

```
wrapper = abs worker
 worker = fix work
```

By applying properties that relate `body`, `abs`, and `rep` the new implementation of `worker` can be optimized. The goal when optimizing `worker`, is to eliminate the intermediate transformations, `rep` and `abs`, from its definition by using the fusion property (Gill & Hutton, 2009). The fusion rule can replace an application of `rep . abs` with an application of `id`, which can be eliminated trivially.

Fusion can only be applied when the `rep` function is composed with the `abs` function. However in many instances, the transformers must somehow be moved through the syntax until they are in a position to apply fusion. In practice, transformers are moved through nodes of an abstract syntax tree using syntax rewrites. For instance, given `body :: A -> A`, the new implementation `work :: B -> B`, and the transformers `rep :: A -> B` and `abs:: B -> A` then the following are true:

```
body . abs = abs . work
```
and
```
rep . body = work . rep
```

And the worker function can be written as

```
worker = fix (rep . abs . work)
```
or
```
worker = fix (work . rep . abs)
```

This changes the syntax, putting the transformers in series. Now, the new implementation is in place and the fusion rule can be applied to eliminate the transformation functions, optimizing `worker`.

The formalized general approach for applying worker/wrapper transformations is outlined in Gill & Hutton (2009). This approach is used to guide the individual function transformations. It dictates the steps necessary to successfully complete a worker/wrapper transformation over a single function. And it was applied using tools capable of performing syntax rewrites, HERMIT.

2.2 The HERMIT System

HERMIT is a framework that provides tools for interacting with and transforming the Haskell Core syntax. The primary means of facilitating this interaction is through the HERMIT Shell, which is a REPL interface that allows the user to traverse a Haskell Core Abstract Syntax Tree (AST). It is inside the shell that the user issues commands that perform AST rewrites (Farmer et al., 2012; Sculthorpe et al., 2013).

HERMIT is capable of manipulating the Core syntax of any Haskell program by suspending its compilation, performing Core AST rewrites, and then allowing compilation to continue. When a worker/wrapper process completes successfully, the binary that is produced maintains the original API but those functions merely exist as wrappers for the new implementation. If desired, HERMIT can expose the worker functions through the API to allow direct access to the new implementation.

The worker/wrapper process is directed by the user. The transformation functions, must be defined by the user prior to the HERMIT session. They are used by HERMIT to apply the worker/wrapper assumptions and split a function into separate worker and wrapper functions.

After a worker/wrapper split, the worker/wrapper methodology is used to apply syntax rewrites in an effort to optimize the worker function and provide the new implementation. The worker definition can be modified using standard HERMIT rewrites or using GHC rules defined with the GHC RULES pragma (Jones et al., 2001), which HERMIT can use to construct new rewrites

(Farmer et al., 2012). These techniques can be used to alter multiple functions in one session, making it possible to change the implementation of an entire program.

2.3 The Game of Life

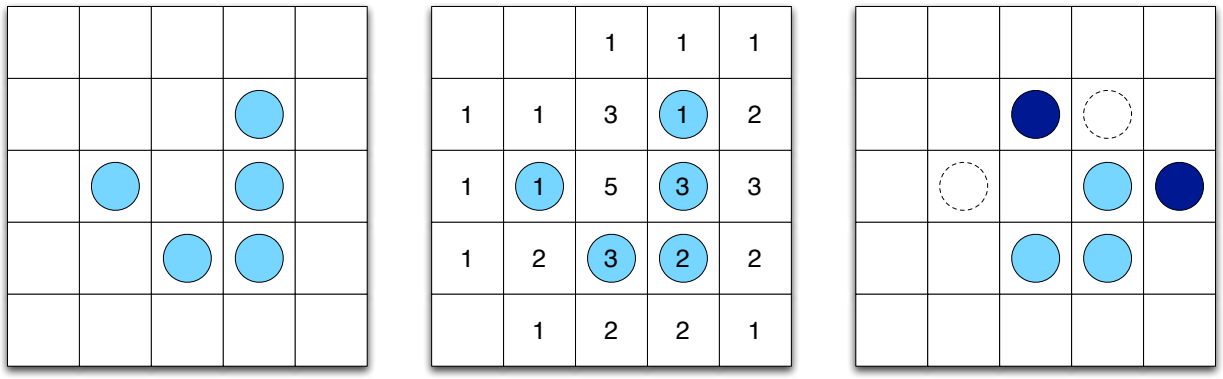
To show that worker/wrapper transformations under HERMIT can be accomplished for programs composed of multiple source files, a suitable program had to be selected that was complex enough to have merit, but not so complex as to provide an overly involved example. The program selected that contains these features is the Game of Life.

The Game of Life was created by the British mathematician John Horton Conway (Gardner, 1970). Technically, it is a simulation. A player creates the initial state of the board and the rules of the game determine how the board evolves, the "player" can only observe once the game has started.

The game board is made of cells arranged in a two dimensional grid. A cell can either be in an alive or dead state. The state of a cell may change depending on the rules of the game. Neighboring cells are simply adjacent cells that share an edge or corner. The rules of the game are simple:

- Under-population – A living cell dies if it has fewer than two living neighbors.
- Over-population – A living cell dies if it has more than three living neighbors.
- Reproduction – A new cell is born if it is empty and has exactly three living neighbors.

The first image in the following figure shows a popular pattern in the game known as the "Glider". The second image shows the same pattern with each cell imprinted with the number of living neighbor cells. These numbers determine the pattern of the next generation, displayed in the third image. The white cells die due to under-population, the dark cells are newly born due to reproduction, and the lightly colored cells remain from the previous generation due to stable-population numbers. These five cells represent the new generation in the game and are used to calculate the next pattern.



Due to the rules of the game, the initial shape will be repeated after four generations. The changes in the pattern through the generations makes the pattern appear to "glide" across the game board in one direction. Many patterns can be produced that "travel", "expand", and even "stabilize", which creates an entertaining display as the generations are propagated, especially if patterns are combined and interact.

Graham Hutton implemented the Game of Life in Haskell using a simple list-based implementation in a terminal console (Hutton, 2007, p. 94-97). The list elements are pairs of integers, `(Int,Int)`. Each pair represents a location on the two-dimensional board structure and has a type synonym, `Pos`. The pairs featured in the list represent the position of living cells in board at a specific generation. Therefore, when a cell dies that position is removed from the list. A position is added to the list when a new cell is born. Hutton's original implementation is the basis of a new implementation of Life that was constructed for these experiments.

The implementation used in this experiment still uses Hutton's original design, however it has been slightly modified. The dimensions of the game board in Hutton's version of Life were hard-coded into the implementation. This version of Life has been modified in a way to allowing the user to alter the board dimensions and also dictate whether or not the edges of the board wrap to create a continuous surface.

Hutton's Life was also divided into two parts. The part of the program that calculates each generation of the game (the engine) has been separated from the part of the program that visualizes each generation (the display). This allows HERMIT to transform the engine without affecting

display functionality.

The board property information is stored into a new type called `Config`. It is simply a type synonym for `((Int,Int),Bool)`. The integer-pair represents the board's size and the boolean value indicates if the board has wrapping edges. The `Config` type and the actual board data, `[Pos]` type are contained in a data structure called `LifeBoard`, that features a constructor and the access functions `board` and `config` which return one of the fields in the data structure. In this implementation of Life it is the `LifeBoard` structure on which most the functions in the program operate. The entire program definition is featured in the Appendix.

Chapter 3

HERMIT Experiments with Life

The goal of this experiment is to attempt the transformation of a complete program module with HERMIT using worker/wrapper methodology, and gather any relevant knowledge in the process.

These experiments were performed using GHC 7.8 in combination with the latest version of HERMIT. All of the examples target the same program described in the previous section, the Game of Life, more specifically, they target the main computation module Life.Engine.Hutton (see Appendix). The transformations take the target module and change the core data structure used in the implementation. This was done by applying the worker/wrapper concept using HERMIT. To properly discuss this process, some terminology needs to be defined. Defining these terms will help make the discussion about these experiments much clearer.

The module that must accompany HERMIT worker/wrapper transformations is referred to as the transformation module. It contains the GHC rules and transformer functions needed for the process.

No special distinction is required when referring to the GHC rules used in this module, but through these experiments, there were distinctions observed between the types of transformation functions.

Transformers are commonly referred to as “rep” or “abs” functions when discussing worker/wrapper theory. And they are always created in pairs (one rep function and one abs function)

called a transformer-pair. Since each operation reverses the operation of the other, they can each be considered as having a polarity that opposes its partner's. This polarity is what allows the fusion property to be applied.

Among transformer pairs there are certain properties that distinguish them from each other. There are two major categories of transformer pairs, atomic and composed. Although a further distinction among the atomic transformer pairs can also be made, simple or compound. Compound atomic transformers are defined using at least one other atomic transformer. Simple atomic transformers are defined independently of any other transformation functions, and typically target the structure desired for conversion. For instance, say there exists a simple atomic rep transformer with the type `rep :: A -> B`. But a function targeted for a worker/wrapper conversion uses the type `[[A]]` in its definition, then two compound atomic rep transformers can be constructed to handle it.

```
repL :: [A] -> [B]
repL xs = map rep xs

repLL :: [[A]] -> [[B]]
repLL xs = map repL xs
```

Definitions for the abs transformers would be similar. Each of these definitions represents examples of compound atomic transformers.

The individual types that make up the type signatures of all simple and compound atomic transformers also reserve special distinction. These types make up a set of types referred to as the "targeted types". Targeted types also come in pairs with a polarity that is inherited by an associated atomic transformer pair. From the example above, the targeted types for these transformers would be the types `A`, `B`, `[A]`, `[B]`, `[[A]]`, and `[[B]]`. Any two types that appear in the same atomic transformer's type signature, are considered opposing types with respect to the worker/wrapper transformation, (i.e. type `A` is the polar opposite of type `B`). Targeted types can also be divided into simple and compound. In this example, `A` and `B` would be simple types, while the rest are compound types. Simple targeted types are converted by simple atomic transformers, and compound targeted

types are converted by compound atomic transformers.

A composed transformer pair is used to split a target function into the worker and the wrapper. They are used to transform functions and are defined using atomic transformers to convert the arguments and/or results of the targeted function. Composed transformers will not have fusion applied to them because they are typically unfolded in the worker optimization process. It is the atomic transformers that are usually moved through the syntax in an effort to apply the fusion property and eliminate them from the final definition.

For instance, carrying on the example from above, lets say a `rep`, `abs`, `repL`, `absL`, `repLL`, and `absLL` functions are defined as atomic transformers, and we wish to target a function with the type `[[A]] -> C -> A` then a composed transformer pair is needed.

```
repLLcA :: ([[A]] -> C -> A) -> [[B]] -> C -> B
repLLcA f bs c = rep (f (absLL bs) c)

absLLcA :: ([[B]] -> C -> B) -> [[A]] -> C -> A
absLLca f as c = abs (f (repLL as) c)
```

This composed transformer pair could be used to split the target function into a worker and a wrapper. Then rewrites would be performed to move the atomic transformers into a position where fusion can be applied.

3.1 Transformation Preparation

Each worker/wrapper conversion using HERMIT requires some preparation before the actual session can begin. The first step is to create a transformation module. For these experiments, each transformation module also contains some useful type synonyms for the `Board` structure (the original type, copied from the source) and the `Board'` structure (the desired type).

```
type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Set Pos)
```

These synonyms are used to increase readability but are not a necessary part of a transformation module. The first experiment converts the main target data type `[Pos]` to the type `Set Pos`, the

Set type is implemented in the Data.Set library. This experiment will be used as an example to guide the discussion.

3.1.1 Transformer Creation

The first thing that the transformation module needs is the set of transformation functions. Staying true to the worker/wrapper methodology, each of the transform names have been designed to begin with "abs" or "rep". Additional characters in the names of the transform functions created in these experiments indicate types associated with the transformation. This naming convention is used throughout this paper for clarity.

Since the goal of the conversion is to change the program's use of `[Pos]` to the use of `Set Pos`, these two types can be considered targeted types of the process. Before constructing any transformers a complete list of targeted types should be constructed to guide atomic transformer creation.

Although the `[Pos]` type is the main target, it is the `Board` type which Life performs most operations. So, this type should be added to the list along with the type of opposing polarity, `Board'`. The types `[Pos]` and `Set Pos` are simple target types, while the `Board` and `Board'` types are considered compound. The targeted types list is complete when all the desired type conversions have been addressed.

Now construction of the atomic transformation pairs can begin. One atomic transformer is required for each of the targeted types. Since simple atomic transformers can be used to define the compound atomic transformers, it is best to start by defining the simple pairs. They are `repb` and `absb` (where the postscript-b refers to a game board). This pair uses predefined functions for list-conversions from the Data.Set library.

```
{-# NOINLINE repb #-}
repb :: [Pos] -> Set Pos
repb = fromList

{-# NOINLINE absb #-}
absb :: Set Pos -> [Pos]
absb = toList
```

The compiler directives shown above these two functions are directions that notify GHC to not automatically inline these functions. GHC will do this automatically for some code in an optimization effort. Without these directives the `absb` and `repb` functions would be unfolded automatically in GHC optimization stages, which complicates the process. Although not required for all the definitions, to avoid this problem, it was found to be best practice to provide this directive for all the transformation functions present in the module.

The simple atomic pair transforms the primary list data structure, but the program mostly operates over the `LifeBoard` structure. The following compound atomic transform pair will be useful for simplifying the overall process.

```
repB :: Board -> Board'
repB b = LifeBoard (config b) $ repb (board b)

absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb (board b)
```

All of the targeted types now have an atomic transformer. Composed transformation pairs must now be added to the transformation module. One composed transformation pair must be created for each of the functions targeted for a worker/wrapper conversion. Consider this composed transform pair used to perform the worker/wrapper split on the `empty` function.

```
repcB :: (Config -> Board) -> Config -> Board'
repcB f c = repB (f c)

abscB :: (Config -> Board') -> Config -> Board
abscB f c = absB (f c)
```

Notice that this pair requires the use of the `repB` and `absB` transformers. Using the atomic transformers to target the arguments and/or results of the targeted function makes constructing the composed transformers straightforward. It also creates uniform and predictable composed transformer definitions. The following pair was created to convert the `alive` function. The type `Scene` is a

type synonym for [Pos], which is the type used by the display mechanism.

```
repBs :: (Board -> Scene) -> Board' -> Scene
repBs f b = f (absB b)

absBs :: (Board' -> Scene) -> Board -> Scene
absBs f b = f (repB b)
```

Comparing the two secondary transformation pairs repcB-abscB and repBs-absBs, a pattern emerges from composed transformer definitions. In places where a target function's result is one of the targeted types, an atomic transformer (with the same polarity as the transformer being defined) is used to convert it. And where the target function's argument is one of the targeted types, an atomic transformer (with the opposite polarity of the transformer being defined) is used for conversion.

For instance, the result of `empty` is the targeted type `Board`, so `repcB` uses `repB` to convert the result of the original function to a `Board'`. But since the argument is a targeted type in `alive`, it uses `absB` to convert the argument before passing it to the original function. This pattern is predictable and uniform and is the basis for a proposed automated composed transformer generator.

The pattern becomes clearer with a function that takes multiple arguments. The next transform pair is used to split the functions `isAlive` and `isEmpty`. Due to their identical type signatures, a single transformer can serve for both conversions.

```
repBpb :: (Board -> Pos -> Bool) -> Board' -> Pos -> Bool
repBpb f b p = f (absB b) p

absBpb :: (Board' -> Pos -> Bool) -> Board -> Pos -> Bool
absBpb f b p = f (repB b) p
```

Only the targeted type `Board` has to be converted in these definitions, but notice how non-target arguments are handled. The `Pos` that is passed to the transformer is simply given to the original function without modification. It is only the targeted types that require special attention in these definitions. The transformer for the `liveneighbs` function has a similar format.

```
repBpi :: (Board -> Pos -> Int) -> Board' -> Pos -> Int
repBpi f b p = f (absB b) p

absBpi :: (Board' -> Pos -> Int) -> Board -> Pos -> Int
absBpi f b p = f (repB b) p
```

In fact, these definitions are identical to the previous definitions. This fact can be used to create polymorphic transformers that serve multiple functions with different type signatures. In addition, due to the capabilities of Haskell the definition could also be shortened to match the definition for the `alive` transformer `repBs`. Then a single definition could be created to handle the worker/wrapper transformation of all four of these functions, like so.

```
repBx :: (Board -> a) -> Board' -> a
repBx f b =  f (absB b)
```

This is a powerful technique that eases the manual creation of these functions, but using polymorphic functions becomes less straightforward and the predictability and uniformity of the composed definitions begin to breakdown. The final two secondary transform pairs required for the transformation module are:

```
reppBB :: (Pos -> Board -> Board) -> Pos -> Board' -> Board'
reppBB f x b = repB (f x (absB b))

abspBB :: (Pos -> Board' -> Board') -> Pos -> Board -> Board
abspBB f x b = absB (f x (repB b))

repBB :: (Board -> Board) -> Board' -> Board'
repBB f b = repB (f (absB b))

absBB :: (Board' -> Board') -> Board -> Board
absBB f b = absB (f (repB b))
```

The first pair is used solely for converting the `inv` function. The second pair is used to complete the transformations of `survivors`, `births`, and `next`, since all three have identical type signatures.

One may notice that the engine module of `Life`, contains the function `neighbs`, but this function does not have an associated transformation. This is because that function is not targeted for a worker/wrapper conversion. Even though this function returns the targeted type `[Pos]`. In this case, it doesn't represent a game board but more literally a list of neighboring positions of a given position, which at most contains eight elements. Converting this function to return `Set Pos` would not improve performance of this function and could actually decrease performance in many cases. These types of situations need to be considered by the user, blindly converting an entire module's

use of a particular type could negatively affect the transformation.

With the transform-pair definitions complete, the transformation module only needs the GHC rules to start the conversion process.

3.1.2 Rule Creation

GHC rules take the form of equality statements given in a GHC RULES pragma. The rules are used to create KURE rewrites that replace syntactic structures with equivalent ones. On the left-side of the equality is the syntax to be replaced, and on the right-side is the syntax desired to take its place. In a transformation module, these rules are applied to the worker function after the original function has been split into a worker and a wrapper. After the initial splitting process, worker functions will be contaminated with intermediate transformations that must be eliminated to optimize the code.

Most of the rules are target specific, they are tailored for a specific function conversion. However, two rules can always be created that take advantage of the fusion property. The first two of the following rules allow users to fuse the transformations and eliminate them from the code. To use these rules, the process will require unfolding the compound atomic transformers, `repB` and `absB`. But doing so will leave a code fragment that HERMIT cannot handle automatically. Unnecessary `LifeBoard` constructors remain when a one of these transformers is unfolded. The last rule was created to eliminate it.

```
{-# RULES
"repb/absb-fusion" [~] forall b. repb (absb b) = b
"absb/repb-fusion" [~] forall b. absb (repb b) = b
"LifeBoard-reduce" [~] forall b. LifeBoard (config b) (board b) = b
 #-}
```

Rules like this may be necessary to reduce some targeted compound types when unnecessary intermediate constructions occur as a result of certain rewrites.

The following rule block contains the rest of the rules needed for the module. They are the rules that use code replacement to eliminate transformations and provide the new implementations within each targeted function.

```
{-# RULES
"empty-l/empty-s" [~]
        repb [] =
        Set.empty
"not-elem/notMember" [~] forall p b.
        not (elem p (absb b)) =
        notMember p b
"elem/member" [~] forall p b.
        elem p (absb b) =
        member p b
"cons/insert" [~] forall p b.
        p : (absb b) =
        absb (insert p b)
"filter/delete" [~] forall p b.
        Prelude.filter ((/=) p) (absb b) =
        absb (delete p b)
"filter-l/filter-s" [~] forall f b.
        Prelude.filter f (absb b) =
        absb (Set.filter f b)
"nub-concatMap/unions" [~] forall f b.
        nub (concatMap f (absb b)) =
        absb (unions (toList (Set.map (fromList . f) b)))
"concat/union" [~] forall b1 b2.
        absb b1 ++ absb b2 =
        absb (union b1 b2)
 #-}
```

Each rule rewrites the worker definition to remove the list-based code and replace it with set-based code. In the process, the rule shifts or eliminates atomic transformers. The most straightforward rule is the one "empty-l/empty-s" rule. The transformation of an empty list results in an empty set. The same result can be achieved by using the empty function from the Data.Set library. HERMIT will use the rule to perform the replacement, eliminating the `repb` function call from the definition.

The rest of the rules operate in a similar fashion, code that uses the type `[Pos]` is changed to use the type `Set Pos`. Most likely, at least one rule will be needed for each target function. But there are cases where none are needed and some cases where multiple rules are needed.

Once all of the transformation functions and rules are in place, the transformation module is complete and the HERMIT session can begin. The user can guide the conversion manually, or HERMIT can be directed with the use of scripts.

3.2 Transformation Process

Whether the HERMIT session is driven with scripts or manually, if an entire module is to be converted, then the order in which the functions are converted must be considered. The order in which the transformations are performed is determined by the dependency relationship of the target functions.

3.2.1 Dependency Issues

The original implementation's functional dependency must be considered prior to the conversion process. If a target function depends on another function that is also targeted for transformation, then this dependency can be transferred to the new implementation by applying the transformations in a particular order.

If the dependency relationships are not considered then the resulting implementation will reach its potential efficiency due to the presence of intermediate transformations. The intermediate transformations occur because the new function implementations may call the wrapper functions instead of the new worker functions directly.

For example, the function `isEmpty` calls the function `isAlive` in the program. And the desire is to maintain this dependent relationship, so the `isAlive` function should be converted before the `isEmpty` function. Converting `isAlive` results in the wrapper `isAlive` and the worker `isAlive'`. This conversion has to be completed first so HERMIT can can connect `isEmpty'` directly to `isAlive'` during its optimization. If the transformation of `isEmpty` is attempted first, then it will call the wrapper `isAlive`. This results in a definition of `isEmpty'` that is inefficient.

Applying an order to the transformations will transfer the dependency relationships and connect the new worker functions to each other and remove the calls to the wrapper functions (and thereby the intermediate transformations).

That being said, there are cases where the dependency relationship is not desired among the worker definitions in the new implementation. In these cases, no special action is necessary, the

replacement code eliminates the dependency and any wrapper calls that may have existed. Therefore, applying the dependency order to the transformation anyway does not affect the resulting program.

For example in this experiment, the function `isEmpty'` will not call the function `isAlive'` in the new implementation. The order in which these functions are converted does not matter. In fact, the transformation of `isEmpty` first unfolds the original `isAlive` to expose the `elem` function in its definition and uses this code as the basis for its replacement rule "not-elem/notMember".

To control the dependency issue in HERMIT, the transformations can be performed manually, or with the use of scripts to control the flow of a program's transformation. This requires knowing which dependency relationships will be transferred to the new implementation. Once these relationships are known, the order of transformation can be determined. In this case, the dependency analysis would have converted the `isAlive` function first, so another approach would have been needed to convert the `isEmpty` function. A rule could be used to replace the call to `isAlive` with a set based function. But since the user has direct control of this aspect, the transformation order can simply be changed.

In these experiments, a master script is used to run each of the other scripts in an order determined by dependency. The use of scripts greatly improves the work-flow of a HERMIT worker/wrapper transformation. HERMIT scripts are simply a series of HERMIT commands that are executed in the given order, so they can be used to ease a program-wide transformation.

3.2.2 The HERMIT Session

Although the transformation process varies between target functions, there are generic steps that can be used to guide any single function transformation.

3.2.2.1 A Transformation Example

The following example will convert the function `inv` and follows the general methods proposed for a worker/wrapper transformation (Gill & Hutton, 2009). The example is illustrated by the

HERMIT terminal output to demonstrate that the goal of the optimization procedure is to eliminate the intermediate transformations.

The `inv` function is used to demonstrate the unique parts of a single worker/wrapper transformation under HERMIT, and is a good example of some of the scenarios that might be encountered. This figure shows the definition of `inv` as the HERMIT session begins.

```
$cinv = λ p b →
  ($) (LifeBoard (config b))
      (case isAlive b p of wild
         False → (:) p (board b)
         True → ($) (filter ((/=) $dEq p)) (board b))
```

The first step is to express the definition as a fixed-point calculation, as required by the worker/wrapper process. In HERMIT, this is done by introducing the `fix` function.

```
$cinv =
  fix (λ $cinv p b →
         ($) (LifeBoard (config b))
             (case isAlive b p of wild
                False → (:) p (board b)
                True → ($) (filter ((/=) $dEq p)) (board b)))
```

The next step is to split the function into the worker and the wrapper. This is done using HERMIT's split command. This command needs the composed transformers given as arguments to properly split the target function (also given as an argument).

```
let worker =
      fix ((.) reppBB
                ((.) (λ $cinv p b →
                        ($) (LifeBoard (config b))
                            (case isAlive b p of wild
                               False → (:) p (board b)
                               True → ($) (filter ((/=) $dEq p)) (board b)))
                     abspBB))
in abspBB worker
```

The worker definition can clearly be seen. The wrapper appears at the `in` clause. Together they perform the same operation as the original `inv` definition. After the split, the next task is to begin

optimizing the new worker function. Our new implementation will use the `isAlive'` function, but it is currently concealed under the wrapper `isAlive`. At this point in all transformations, any wrapper functions that are present should be unfolded to expose the worker function.

```
fix ((.) reppBB
          ((.) (λ $cinv p b →
                   ($) (LifeBoard (config b))
                       (case absBpb isAlive' b p of wild
                          False → (:) p (board b)
                          True → ($) (filter ((/=) $dEq p)) (board b)))
               abspBB))
```

Notice that unfolding wrappers not only exposes the worker, but it also exposes the composed "abs" transformer that is a part of every wrapper's definition. When all the wrappers have been unfolded, then all of the composed transformers should be unfolded to gain access to the atomic transformers, namely the functions `absBpb`, `reppBB`, and `abspBB` in this case.

```
fix ((.) (λ f x b → repB (f x (absB b)))
          ((.) (λ $cinv p b →
                   ($) (LifeBoard (config b))
                       (case isAlive' (repB b) p of wild
                          False → (:) p (board b)
                          True → ($) (filter ((/=) $dEq p)) (board b)))
               (λ f x b → absB (f x (repB b)))))
```

Now at this point, it would be perfectly valid to attempt to fuse the compound atomic transformations, but the rules created for this example focus on the simple atomic transformers. So the next step is to unfold the compound atomic transformers to expose the simple atomic transformers.

```
fix ((.) (λ f x b →
            LifeBoard (case f x
                           (LifeBoard (case b of wild
                                          LifeBoard ds1 ds2 → ds1)
                                      (case b of wild
                                          LifeBoard ds1 ds2 → absb ds2))
                       of wild
                        LifeBoard ds1 ds2 → ds1)
                      (case f x
                           (LifeBoard (case b of wild
                                          LifeBoard ds1 ds2 → ds1)
                                      (case b of wild
                                          LifeBoard ds1 ds2 → absb ds2))
                       of wild
                        LifeBoard ds1 ds2 → repb ds2))
         ((.) (λ $cinv p b →
                 ($) (LifeBoard (config b))
                     (case isAlive' (LifeBoard (case b of wild
                                                    LifeBoard ds1 ds2 → ds1)
                                               (case b of wild
                                                  LifeBoard ds1 ds2 → repb ds2))
                                    p
                      of wild
                       False → (:) p (board b)
                       True → ($) (filter ((/=) $dEq p)) (board b)))
              (λ f x b →
                 LifeBoard (case f x
                                (LifeBoard (case b of wild
                                               LifeBoard ds1 ds2 → ds1)
                                           (case b of wild
                                              LifeBoard ds1 ds2 → repb ds2))
                             of wild
                              LifeBoard ds1 ds2 → ds1)
                           (case f x
                                (LifeBoard (case b of wild
                                               LifeBoard ds1 ds2 → ds1)
                                           (case b of wild
                                              LifeBoard ds1 ds2 → repb ds2))
                             of wild
                              LifeBoard ds1 ds2 → absb ds2))))
```

Unfolding the compound transformers (`repB` or `absB`) causes a `LifeBoard` constructor to appear. These fragments can be eliminated later during optimization. Now, HERMIT's simplify command should be applied to reduce the current syntax. However, the definitions of the `config` and `board` functions should also be unfolded to eliminate the transformers from their definitions. In addition, the fixed-point calculation can also be eliminated by simply unfolding it prior to the simplify command. The next figure shows the syntax with these functions also unfolded.

```
let rec x =
          (.) (λ f x b →
                 LifeBoard (case f x
                                   (LifeBoard (case b of wild
                                                  LifeBoard ds1 ds2 → ds1)
                                              (case b of wild
                                                 LifeBoard ds1 ds2 → absb ds2))
                                of wild
                                 LifeBoard ds1 ds2 → ds1)
                               (case f x
                                       (LifeBoard (case b of wild
                                                      LifeBoard ds1 ds2 → ds1)
                                                  (case b of wild
                                                     LifeBoard ds1 ds2 → absb ds2))
                                of wild
                                 LifeBoard ds1 ds2 → repb ds2))
              ((.) (λ $cinv p b →
                      ($) (LifeBoard (case b of wild
                                             LifeBoard ds1 ds2 → ds1))
                          (case isAlive' (LifeBoard (case b of wild
                                                           LifeBoard ds1 ds2 → ds1)
                                                    (case b of wild
                                                       LifeBoard ds1 ds2 →
                                                         repb ds2))
                                         p
                           of wild
                            False →
                              (:) p
                                  (case b of wild
                                     LifeBoard ds1 ds2 → ds2)
                            True →
                              ($) (filter ((/=) $dEq p))
                                  (case b of wild
                                     LifeBoard ds1 ds2 → ds2)))
                   (λ f x b →
                      LifeBoard (case f x
                                      (LifeBoard (case b of wild
                                                     LifeBoard ds1 ds2 → ds1)
                                                 (case b of wild
                                                    LifeBoard ds1 ds2 →
                                                      repb ds2))
                                   of wild
                                    LifeBoard ds1 ds2 → ds1)
                                  (case f x
                                          (LifeBoard (case b of wild
                                                         LifeBoard ds1 ds2 → ds1)
                                                     (case b of wild
                                                        LifeBoard ds1 ds2 →
                                                          repb ds2))
                                   of wild
                                    LifeBoard ds1 ds2 → absb ds2)))
              x
in x
```

The code has become almost unmanageable, luckily it is time to use the simplify command. This

command is actually a small series of rewrites that reduces code to a very compact form. Observe the difference in code size compared to the simplified code.

```
λ x b →
  LifeBoard (case b of wild
                LifeBoard ds1 ds2 → ds1)
            (repb (case isAlive' (LifeBoard (case b of wild
                                                LifeBoard ds1 ds2 → ds1)
                                            (repb (case b of wild
                                                      LifeBoard ds1 ds2 →
                                                        absb ds2)))
                                 x
                        of wild
                         False →
                           (:) x
                               (case b of wild
                                  LifeBoard ds1 ds2 → absb ds2)
                         True →
                           filter ((/=) $dEq x)
                                  (case b of wild
                                     LifeBoard ds1 ds2 → absb ds2)))
```

The fixed-point calculation has been eliminated in the process. And the unnecessary transformations associated with the `config` function calls has also been eliminated. Now, all the instances of `config` can safely be folded again.

```
λ x b →
  LifeBoard (config b)
            (repb (case isAlive' (LifeBoard (config b)
                                            (repb (case b of wild
                                                      LifeBoard ds1 ds2 →
                                                        absb ds2)))
                                 x
                        of wild
                         False →
                           (:) x
                               (case b of wild
                                  LifeBoard ds1 ds2 → absb ds2)
                         True →
                           filter ((/=) $dEq x)
                                  (case b of wild
                                     LifeBoard ds1 ds2 → absb ds2)))
```

The same type of operation needs to occur for the `board` function calls, but an extra step is required. The functions `config` and `board` are reduced to case statements in Core syntax.

That allows HERMIT to use a "case-float" rewrite to move certain operations outside of the case statement. The `absb` functions need to be lifted out of the `board` definition and this command is used to do that.

```
λ x b →
  LifeBoard (config b)
            (repb (case isAlive' (LifeBoard (config b)
                                            (repb (absb (case b of wild
                                                                  LifeBoard ds1 ds2 →
                                                                    ds2))))
                                 x
                        of wild
                         False →
                           (:) x
                               (absb (case b of wild
                                               LifeBoard ds1 ds2 → ds2))
                         True →
                           filter ((/=) $dEq x)
                                  (absb (case b of wild
                                                   LifeBoard ds1 ds2 → ds2))))
```

Notice how the `absb` functions floated out the three case statements. Now that they have been moved, the `board` function calls can be folded once again.

```
λ x b →
  LifeBoard (config b)
            (repb (case isAlive' (LifeBoard (config b) (repb (absb (board b))))
                                 x
                        of wild
                         False → (:) x (absb (board b))
                         True → filter ((/=) $dEq x) (absb (board b))))
```

Now the code has been substantially reduced and the atomic transformers have been moved into positions where rules can be applied. The process could begin by fusing some of the transformers, but it would be more efficient if they were all fused at the same time. The goal now is to get all of the transformations ready for fusion.

This is made slightly more difficult by the fact that this function contains a branch (exhibited by the two alternates of the `case` statement). Each of these branches contains an atomic transformer that must be eliminated, but there is only one `repb` transformer left to fuse them. They will have to be merged as they are brought out of the branching expression. That can be accomplished with

another "case-float" command, but each branch must begin with the same transformer for that to work. Each must have its code replaced with a new implementation. The rules that do this for each branch also move the transformers into position. When the code is replaced in both branches using the rules cons/insert and filter/delete featured in the transformation module, the resulting definition is:

```
λ x b →
  LifeBoard (config b)
            (repb (case isAlive' (LifeBoard (config b) (repb (absb (board b))))
                                 x
                    of wild
                     False → absb (insert $dOrd x (board b))
                     True → absb (delete $dOrd x (board b))))
```

Each branch now begins with the same function application and the `case` statement can be rewritten using a "case-float" command.

```
λ x b →
  LifeBoard (config b)
            (repb (absb (case isAlive' (LifeBoard (config b)
                                                  (repb (absb (board b))))
                                       x
                        of wild
                         False → insert $dOrd x (board b)
                         True → delete $dOrd x (board b))))
```

Now all the transformers are ready for fusion. The next figure shows the definition with the transformers eliminated. But notice that there is a `LifeBoard` construction that doesn't need to occur.

```
λ x b →
  LifeBoard (config b)
            (case isAlive' (LifeBoard (config b) (board b)) x of wild
               False → insert $dOrd x (board b)
               True → delete $dOrd x (board b))
```

The LifeBoard-reduce rule is needed here. The unnecessary constructor appears from the unfolding of one of the compound atomic transformers, `repB` or `absB`. After fusion, the transformers are gone but the constructor remains. Applying the LifeBoard-reduce rule brings the definition to its optimal state.

```
λ x b →
  LifeBoard (config b)
            (case isAlive' b x of wild
               False → insert $dOrd x (board b)
               True → delete $dOrd x (board b))
```

With the optimization of the worker completed, all that remains is to do is to rename the worker and move it to the top of the syntax tree so that other worker functions will have direct access to it.

When a similar process has been performed for all of the module or program functions, then the program transformation is complete. Compilation under GHC is resumed with the modified Core syntax to produce the final product.

3.3 An Alternate Approach

There is a slightly simpler process for the worker/wrapper transformations regarding individual function transformations. The previous method chose to unfold compound atomic transformers to expose simple atomic transformers. However, the fusion property can be applied to compound atomic transformers as well. Using this alternate method reduces the number of rewrites needed without sacrificing confidence in the transformations. Basically, some rules are created that combine several rewrites into one. The same transformation functions can be used in this method, but some changes to the rules in the transformation module are required. The repb/absb-fusion rule will be replaced by a similar repB/absB-fusion rule that fuses the compound transformers.

The previous method unfolded the `config` and `board` functions so the transformers could be handled. In the case of `config` this eliminates the transformer. And in the case of `board` another rewrite is required to move the function out of its definition. These steps can be accomplished with

a single rewrite rule for each case.

```
{-# RULES
"repB/absB-fusion" [~] forall b. repB (absB b) = b
"config-absB" [~] forall b. config (absB b) = config b
"board-absB" [~] forall b. board (absB b) = absb (board b)
"LifeBoard-absB" [~] forall c b.
        LifeBoard c (absb b) =
        absB (LifeBoard c b)
 #-}
```

Rules like "board-absB" simultaneously shifts the transformation and reduces it to a simple transformer. Since this method focuses on fusing the compound transformers, then another rule "LifeBoard-absB" is used to simultaneously promote atomic transformers and shift the transformation.

The rest of the rules are very similar to the previous method's rules for replacing code. The only differences are that these rules are defined with compound transformers rather than the simple transformers.

The start of the process is the same as in the previous method. The function has a fixed-point calculation introduced, is split, and then the wrappers and composed transformers are unfolded (along with the `fix` function), all before it is simplified. The atomic transformers are left in place. The next figure shows the state of the syntax after simplification.

```
λ x b →
  repB (LifeBoard (config (absB b))
                  (case isAlive' (repB (absB b)) x of wild
                     False → (:) x (board (absB b))
                     True → filter ((/=) $dEq x) (board (absB b))))
```

With this approach, the goal is to fuse any of the atomic transformer pairs. The same case-float trick can be used to merge the transformers in each branch, but first the board-absB rule needs to be applied to shift the transformation out of the `board` function.

```
λ x b →
  repB (LifeBoard (config (absB b))
                  (case isAlive' (repB (absB b)) x of wild
                     False → (:) x (absb (board b))
                     True → filter ((/=) $dEq x) (absb (board b))))
```

Now the same "cons/insert" and "filter/delete" rules can be applied to the branches to replace the

list-based code. Then another "case-float" rewrite can be used to merge the absb functions. The following code is the result of those actions.

```
λ x b →
  repB (LifeBoard (config (absB b))
                  (absb (case isAlive' (repB (absB b)) x of wild
                              False → insert $dOrd x (board b)
                              True → delete $dOrd x (board b))))
```

By applying the new LifeBoard-absb rule, the simple transformer is promoted to a compound transformer and shifted outside of the constructor.

```
λ x b →
  repB (absB (LifeBoard (config b)
                        (case isAlive' (repB (absB b)) x of wild
                           False → insert $dOrd x (board b)
                           True → delete $dOrd x (board b))))
```

And finally, with the application of fusion, the definition is optimized and matches the result of the previous method.

```
λ x b →
  LifeBoard (config b)
            (case isAlive' b x of wild
               False → insert $dOrd x (board b)
               True → delete $dOrd x (board b))
```

The method shown here is easier because there are more choices involved when all the atomic transformers are accessible during optimization rather than only simple atomic transformers. The ability to go between simple atomic and compound atomic transformations eases the process. This becomes especially useful if the transformation is focused on changing the program's basic representation of data. The preparation process can become slightly more complicated in these cases, so any simplification in the transformation itself is welcome.

3.4 Representation Transformations

Transforming Life to use sets instead of lists was not the only successful experiment. The program was also converted to use the `QuadTree` structure from the `Data.QuadTree` library, as well as the `UVector` implemented in the `Data.Vector.Unboxed` library.

The unique aspect of these two transformation experiments is that along with the transformation of the actual structure used in the program, the representation of the data was also transformed. These experiments showed that not only could the implementation change but the representation of data could also be altered in the process. The original version of Life uses `[Pos]` to contain the coordinates of living cells. These transformations replace this representation with one that constantly represents every cell on the board as a boolean value. Occupied cells receive the value `true` and empty cells receive the value `false`.

To properly construct a quad-tree or vector of boolean values, the size of the board had to be known. The board's size data is contained in the `Config` type which is a part of the `LifeBoard` structure. The information is readily available to compound atomic transformers over this type, but this data is not available to the simple atomic structures that operate on the type `[Pos]`. The only way that these transformers can have access to that data is to receive it as an argument. This segment shows the simple atomic definition for the quad-tree transformation.

```
indices :: Size -> [Pos]
indices (w,h) = [ (x,y) | x<-[0..w-1], y<-[0..h-1] ]

repb :: Size -> [Pos] -> QuadTree Bool
repb sz = foldr (\p qt -> setLocation p qt True) $ makeTree sz False

absb :: QuadTree Bool -> [Pos]
absb qt = [ p | p <- indices (treeDimensions qt), getLocation p qt ]
```

The `indices` function was provided as a helper simply to construct a list of all the board positions. It makes the definition of `absb` simpler. Notice, that it is only the `repb` function that requires an extra argument. Since the quad-tree carries the board size data with it, the `absb` transformer has access to this data through the quad-tree structure passed to it. This is in contrast to the vector transformation experiment where both of the simple atomic transformers required extra arguments.

```
repb :: Size -> [Pos] -> Vector Bool
repb sz b =
    generate ((fst sz)*(snd sz))
             (\i -> Prelude.elem (i `mod` (fst sz), i `div` (fst sz)) b)

absb :: Size -> Vector Bool -> [Pos]
absb sz b = [ (x,y) | x <- [0..((fst sz) - 1)],
                      y <- [0..((snd sz) - 1)],
                      b ! (y * (fst sz) + x) ]
```

Compared to the structure conversion in the set transformation example these simple atomic transformations have more complex definitions due to altering the data representation. The extra complexity effects the definitions of the compound atomic transformations. The following are used in the quad-tree conversion.

```
repB :: Board -> Board'
repB b = LifeBoard c $ repb (fst c) $ board b
        where c = config b

absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb $ board b
```

However, the compound transformers conceal the extra complexity. If fusion efforts are focused on any atomic transformer (as opposed to just the simple atomic transformers), then the process is somewhat simplified because they can be eliminated without exposing the simple atomic transformations. Overall the only real complications that come from representation transformations is a slight increase in complexity of the atomic transformer definitions. Complications are not added to the overall process within HERMIT. The general worker/wrapper methods can still be applied safely.

Chapter 4

Discussion

The experiment was successful in two ways. One obvious success is that this is the first time that an entire program has been transformed using the worker/wrapper methods. This builds on previous successes using HERMIT to transform individual functions and progresses the study to the next step in attempting larger-scale transformations. However, a less obvious success is that the observations establish patterns across all transformations. These patterns can be used to improve the worker/wrapper process in HERMIT. An effort to improve HERMIT is already underway, but these experiments have uncovered a pattern that exposes the potential for automating the worker/wrapper process even further.

4.1 The Worker/Wrapper Transformation Recipe

A general recipe to perform a worker/wrapper transformation on an individual function is established in (Gill & Hutton, 2009). By expanding the original recipe, to consider multiple-function transformations, the process can be interwoven with recipes for module and program transformations.

The recipes shown here can be used to perform individual function transformations or program transformations. The same basic steps originally outlined, have not been altered but certain steps have been added to include some beneficial steps. The recipe is also refined slightly for use with

the program transformation recipe proposed. It has been broken into two stages the preparation and procedural stages.

The following shows a simple list of the generic steps to prepare for the successful worker/wrapper transformation of a single function.

1. Select the target types and match each with the type to which it will be converted.
2. Define the simple atomic transformer pairs needed to convert the target types. Then construct the compound atomic transformer pairs from the simple atomic transformers.
3. Define the composed transformer using the definitions of the atomic transformers to convert the target function's arguments and/or result.
4. Ensure that each transformer pair satisfies a worker/wrapper assumption.

The following steps can be followed to perform an individual function transformation after preparation has occurred:

1. Rewrite the target function's definition as a least-fixed-point calculation.
2. Split the target function, creating the worker and wrapper functions.
3. In the worker definition, unfold any wrapper functions or composed transformers.
4. Apply rewrites needed to eliminate the atomic transformers.
5. Apply rewrites to optimize the worker, including the removal of the fixed-point calculation.

In addition, the following recipe can guarantee the successful worker/wrapper transformation of an entire module, by expanding these steps to consider multiple functions and multiple target types.

1. Select all of the functions targeted for transformation.
2. Repeat the preparation stage for all of the target functions.
3. Construct a dependency graph of the module's functions and use it to determine the order of the transformations.
4. Repeat the transformation stage for each of the targeted functions in the order determined by their dependencies.

The steps to convert an entire program are an extension of the module transformation recipe.

1. Select all of the modules targeted for transformation.
2. Construct a dependency graph of the module relationships and use it to determine the order of the module transformations.
3. Repeat the transformation process for each of the targeted modules in the order determined by their dependencies.

These recipes can be used to perform successful worker/wrapper transformations of entire programs using any tool capable of performing the necessary operations. These experiments specifically explored the use of the HERMIT tool with this methodology. In the process, observations were made that could be used to improve the its worker/wrapper capabilities.

4.2 HERMIT's Worker/Wrapper Process

This section introduces some ideas that could be used to improve the worker/wrapper process within HERMIT, including additional features that would reduce the preparation stages of the process.

One of the attractions of being able to perform these transformations is the idea that one does not have to change the original source code to provide an improved implementation. However for a worker/wrapper transformation in HERMIT, the module targeted for the conversion must import the transformation module, which requires editing the source for each transformation module. This is done so when the original module is targeted in HERMIT, the system will have access to the transformation functions and GHC rules provided.

This dependency should be eliminated (or at least reversed so that the transformation module imports the targeted module). This would require changes to the linking methods used by HERMIT. Perhaps, even a feature could be added to HERMIT that allows the transformation module to be linked into a live session. If HERMIT could do this then the transformation module could

be linked into the compilation when the HERMIT session begins, avoiding the need to modify the original source.

Another improvement to the HERMIT's worker/wrapper transformation process involves combining several HERMIT commands into more robust commands that perform the actions of several. The process to convert a single function has steps that are uniform across all transformations. These steps occur at the beginning and end of the transformation process.

Once the worker function has been optimized, the steps required to finalize the transformation are always the same. The worker code is first renamed and moved to the top of the syntax tree. These steps could be combined into a single command. The user would then not have to explicitly perform these actions. This would simplify the cleanup process needed after worker optimization.

Another improvement could be applied to HERMIT's use of the split command. Under HERMIT, the command only works on functions that include the least fixed-point calculation. Typically, this is accomplished by introducing the `fix` function prior to attempting the worker/wrapper split. The introduction of `fix` could easily be folded into the actions of the split command so that the user does not have to explicitly perform this operation. Furthermore, the navigation steps required to get to the function's definition could also be included as part of the command's operation using a top-down search of the syntax tree.

These steps are required to perform the split command, so a single command could be made to find the given function, perform the necessary navigation, introduce the `fix` function, perform the worker/wrapper split, navigate to the worker function, and then immediately unfold any wrappers and composed transformers. Providing all of these actions under a single command would improve the work-flow for the process.

In addition, the current implementation of the split command requires that the transformation functions be given as arguments to the command, for example "split-1-beta births [|absBB|] [|repBB|]". But this could be improved to increase work-flow. The transformation module contains all the relevant information needed to further automate the transformation process, but it would require additional features in HERMIT. If the transformation module were analyzed when it

is loaded, it can provide much useful information about the intended transformation. If HERMIT scanned the atomic transformer definitions, it could determine all of the targeted types and associate them with the proper transformers. And if HERMIT could maintain a record of this data, then the user would not have to explicitly indicate the transformers to use with the split command. For example, when the "split" command is given, the target function's type signature could be scanned and the associated composed transformer pair could be found easily by comparing the type of the function argument of the compound transformers. When a match is found, that is the transformer needed and since HERMIT would also keep a record of which transformers have opposing polarity, the appropriate pair could be found automatically.

This improvement would further be supported by an algorithm to automatically generate the necessary composed transformers when the "split" command is given. Composed transformers could be constructed on-call given a properly defined set of atomic transformers. This would eliminate the need for the users to define the composed transformers explicitly in the transformation module. The next section describes a suggested algorithm for creating composed transformation pairs on-the-fly. Adding such an algorithm would eliminate the need for a type matching system just described because the functions could be generated from the information gleaned from the target function type and the atomic transformers.

Whether using transformer generation or code analysis were added to HERMIT, then the split command would only require the target functions name. It would be able to complete the common initial transformation steps with the information provided in the transformation module. However, with the implementation of a composed transformer generator, the user would never have to explicitly define the composed transformers. This would greatly shorten the preparation stage of the process.

4.2.1 Automating Transformer Creation

The inclusion of an automated composed transformer generator to HERMIT would first require some additional features be added to the system. HERMIT would be required to actively process

the transformation module to gather crucial data.

It would need the ability to scan the transformation module to gather data. The process would start with this analysis phase. This would primarily be to build a list of the targeted types. The type's used in the atomic transformers indicate which types the user is interested. Also by analyzing the atomic transformers, each target type can have a transformer associated with it. Then whenever that type is encountered, HERMIT will know which atomic transformation to apply. Each target type can also be associated with a target type of opposite polarity because this information is also indicated by the types used in the atomic transformers.

If the polarity and transformation relationship data is maintained in HERMIT, then when the user invokes a worker/wrapper split command on a function, HERMIT could generate the composed transformer pair needed rather than requiring its definition in the transformation module.

4.2.2 Further Automation Solutions

One simple addition to HERMIT's worker/wrapper protocol would be the automatic creation of transformer fusion rules. Each of the atomic transformer pairs (if defined properly) will have the inherent property which states that `rep . abs = id` and `abs . rep = id`. Therefore, fusion rules could automatically be generated for each atomic transformer pair. This addition doesn't save the user much time but it would give HERMIT the ability to apply these rules automatically to eliminate transformations without user input.

Improvements to the worker/wrapper commands along with the addition of targeted type analysis, a composed transformer generator, and dependency relationship analysis would greatly improve HERMIT's worker/wrapper ability by reducing the effort required by the user. With HERMIT's current features, these efforts could easily become overwhelming if large-scale programs are targeted for conversion.

The desire exists for an automated worker/wrapper transformation process, but because the syntax rewrites required to eliminate the transformers and optimize the worker vary between targets, this currently cannot be accomplished. From this experiment, observations were made that indi-

cate special combinations of rewrites might provide predictable transformer movement through the syntax tree. But the nature of worker/wrapper transformations must be studied further to determine if these indicators have merit. If there are rewrites that can predictably move arbitrary transformers through the syntax, then this feature could be used in conjunction with fusion to automatically eliminate transformations, reducing user responsibility which would be a welcome enhancement to the process.

Chapter 5

Related Work

HERMIT inherits its rewrite capability from the embedded Haskell DSL, KURE. It stores Haskell's Core syntax as a strongly-typed syntax tree and provides advanced techniques to alter it. A similar system was designed to explore program transformation called Stratego (Bravenboer et al., 2008). However, this system was designed to use a declarative-style intermediate language and cannot be used to transform Haskell programs.

Similar rewrite systems have been created to explore language transformation benefits in Haskell. The most notable being HaRe (Brown, 2008), and PATH (Tullsen, 2002). These systems also transform the Haskell language. However, these systems were designed to operate on a subset of Haskell. PATH operates on its own version of a Haskell-like syntax, and HaRe only operates on the Haskell98 subset. Only HERMIT maintains a close relationship with the industrial version of GHC, which makes it an ideal candidate for studying large-scale, real-world program transformations.

Earlier work, in Farmer et al. (2012); Sculthorpe et al. (2013), showed that worker/wrapper can be successfully mechanized in the small. The experiments presented in this thesis extend that work to include the transformation of an entire program. In addition, to changing a programs structure and representation. Successful experiments in changing a program's method of computation have also been conducted, this is accomplished by leveraging a GPGPU.

Efforts at doing this have targeted the Accelerate DSL, which makes it possible to program GPGPUs using Haskell (Chakravarty et al., 2011; McDonell et al., 2013). The experiment succeeded at transforming the implementation to perform the program's computations on the resident GPGPU, demonstrating that worker/wrapper transformations can be extended to changing the hardware on which the program is performed.

The HERMIT toolkit has experienced success in a variety of applications. Some of those applications include applying the worker-wrapper transformation to optimize functions like `reverse` and `last` Farmer et al. (2012); Sculthorpe et al. (2013). Additionally, HERMIT has also been used to mechanize an optimization pass for SYB Adams et al. (2014), and to enable stream fusion for `concatMap` Farmer et al. (2014).

Chapter 6

Conclusion

The original version of Life uses `[Pos]` to contain the coordinates of living cells which is an efficient representation of the board if it is sparsely populated. But when the board is densely populated this structure begins to see a loss in performance due to long access times associated with walking a linked list. Each of the transformations presented also attempted to improve program performance by decreasing the execution time of the engine module.

6.1 Life Performance Results

Three transformations of Life using the set, vector, and quad-tree data structures as targets were put into a testing framework and tested for correctness and performance. Correctness tests were performed using the Test.QuickCheck library. The tests were performed on various board sizes with varying initial conditions for 1000 runs each and the results were compared to the results of the original implementation.

Each transformation results in a set of programs that each use a different data structure from the original. To accurately test for correctness, the test results had to be converted to the same type. The `alive` function is used to accomplish this. But since the original board representation does not keep the list of living positions sorted, the results of these test runs had to be sorted before comparison.

All of the transformations produced the same resulting generation, demonstrating that the transformations are indeed performing the same computations and producing the same output.

6.1.1 Performance Evaluation

Although not the primary goal of the experiment, the performances of each implementation were compared for observation. These tests were performed using the Criterion.Main library. Tests were run for board sizes ranging from small to large with various initial conditions. The tests timed the implementations for 1,000,000 generations. The tests were designed to compare the performance of the program's engine module, and the display portions of the program remained disconnected.

The results of the HERMIT transformations were compared against the original Life engine compiled using just GHC. The following table shows performance test results for two tests, one uses a small board initialized with a single Glider pattern, the other uses a larger board initialized with a large formation of Gliders. Both tests use a wrapping board structure so the pattern can repeat infinitely.

Implementation	20x20	160x160
List	112.86 ms	125.74 ms
Set	77.85 ms	76.71 ms
Quad-Tree	78.41 ms	76.95 ms
Unboxed Vector	77.20 ms	76.46 ms

Compared to the original implementation, all of the HERMIT transformations showed a 30-40% improvement in execution time over the original implementation. These results show the tendency for more populated boards to slow the list implementation. But this does not affect the transformed implementations since those data structures have faster access times. In fact, the overhead associated with these structures is mitigated with increased board density, which results in heavily populated simulations having faster execution times than the more sparsely populated simulations.

However, it should be noted that when compared to the original engine compiled with GHC using the "-O2" optimization flag, the original implementation is optimized enough to increase performance substantially. There was no notable improvement or loss in performance from transformations when the original program was compiled using GHC optimization. All cases executed in relatively the same amount of time.

6.2 Denouement

This thesis has demonstrated the worker/wrapper transformation process using HERMIT to manipulate the Game of Life programmed in Haskell. Through these experiments observations were made that can improve the HERMIT tool as well as increase knowledge about the worker/wrapper transformation process.

The proposed improvements to HERMIT are meant to decrease the effort of the user. The current worker/wrapper transformation process would become overwhelming for large-scale programs. The fact that every targeted type requires a transformer and each targeted function requires two transformers, leads one to imagine the number transformers needed to convert a program that has hundreds of function definitions. Presently, it simply would not be practical to attempt a large-scale worker/wrapper transformation using HERMIT. But through experiments like this one, observations about the process and the tools can be made to help improve the collective knowledge. Improving HERMIT would allow larger experiments to be attempted and more knowledge to be gained in the process.

Adding a record-keeping feature to track the targeted types and their related transformers could vastly improve HERMIT's transformation process. Making this knowledge available to HERMIT would allow the implementation of a composed transformer generator, which reduces the user responsibility, increasing work-flow. Additionally, adding dependency relationship analysis capabilities to HERMIT could determine the proper order for transforming the functions of a module as well as the order for transforming modules in a program.

The performance tests showed an increase in program performance using these techniques. Demonstrating that worker/wrapper transformations can be used to improve a program. Not only did these experiments successfully improve program performance, but these experiments have led to discoveries that may further mechanize the worker/wrapper process, increasing HERMIT's capability. Finally, the observations culminated in a generalized recipe for module and program scale worker/wrapper transformations.

References

Adams, M. D., Farmer, A., & Magalhães, J. P. (2014). Optimizing SYB is Easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14 (pp. 71–82). New York, NY, USA: ACM.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. (2008). Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1–2), 52–70.

Brown, C. M. (2008). *Tool Support for Refactoring Functional Programs*. PhD thesis, University of Kent.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., & Grover, V. (2011). Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (pp. 3–14).: ACM.

Farmer, A., Gill, A., Komp, E., & Sculthorpe, N. (2012). The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proceedings of the ACM SIGPLAN Haskell Symposium*, Haskell '12 (pp. 1–12).: ACM.

Farmer, A., Höner zu Siederdissen, C., & Gill, A. (2014). The HERMIT in the Stream: Fusing Stream Fusion's concatMap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14 (pp. 97–108). New York, NY, USA: ACM.

Gardner, M. (1970). Mathematical Games – The Fantastic Combinators of John Conway's New Solitaire Game "Life". *Scientific American*, 223, 120–123.

Gill, A. & Hutton, G. (2009). The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(02), 227–251.

Hutton, G. (2007). *Programming in Haskell*. Cambridge University Press.

Jones, S. P., Tolmach, A., & Hoare, T. (2001). Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell Workshop*, volume 1 (pp. 203–233).

McDonell, T. L., Chakravarty, M. M., Keller, G., & Lippmeier, B. (2013). Optimising Purely Functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming* (pp. 49–60).: ACM.

Sculthorpe, N., Farmer, A., & Gill, A. (2013). The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science* (pp. 86–103).

Sculthorpe, N. & Hutton, G. (2014). Work It, Wrap It, Fix It, Fold It. *Journal of Functional Programming*, 24(1), 113–127.

Tullsen, M. (2002). *A Program Transformation System for Haskell*. PhD thesis, Yale University.

Appendix A

Appendix of Code

The code featured here shows the Haskell implementation of the Game of Life that was created for these experiments. Also included, is the transformation module for the set and quad-tree transformation experiments.

A.1 The Game of Life

The Game of Life Implementation modules featured here are:

1. `Life.Types` Defines the base-types used in the program
2. `Life.Formations` Contains predefined initialization data
3. `Life.Engine.Hutton` Defines the original computation module, which is the target of the transformations.
4. `Life.Display.Console` Defines the terminal display module
5. `Main` Defines an example program that merges the engine and display implementations into an operational program, that runs a predetermined board structure.

A.1.1 Life.Types

```
module Life.Types where

import Data.List (nub)

type Size = (Int,Int)
type Config = (Size,Bool)
type Pos = (Int,Int)
type Scene = [Pos]

data LifeBoard c b = LifeBoard
        { config :: c
        , board :: b }
    deriving Show

class Life b where
    empty :: Config -> b
    alive :: b -> Scene
    inv :: Pos -> b -> b
    next :: b -> b

scenes :: [Scene] -> Scene
scenes s = nub $ concat s

scene :: Life board => Config -> Scene -> board
scene c@((w,h),warp) =
    foldr inv (empty c) . (if warp
        then map (\(x,y) -> (x `mod` w, y `mod` h))
        else filter (\(x,y) -> (x >= 0 && x < w) && (y >= 0 && y < h)))

runLife :: Life board => Int -> board -> board
runLife 0 b = b
runLife n b = runLife (n-1) (next b)
```

A.1.2 Life.Formations

```
module Life.Formations where

import Life.Types

glider :: Pos -> Scene
glider (x,y) = [(x+2,y+3),(x+3,y+4),(x+4,y+2),(x+4,y+3),(x+4,y+4)]

gliders :: Pos -> Scene
gliders (x,y) = scenes [glider (x,y), glider (x+5,y),
                        glider (x,y+5), glider (x+5,y+5)]

gliders2 :: Pos -> Scene
gliders2 (x,y) = scenes [gliders (x,y), gliders (x+10,y),
                         gliders (x,y+10), gliders (x+10,y+10)]

gliders3 :: Pos -> Scene
gliders3 (x,y) = scenes [gliders2 (x,y), gliders2 (x+20,y),
                         gliders2 (x,y+20), gliders2 (x+20,y+20)]

gliders4 :: Pos -> Scene
gliders4 (x,y) = scenes [gliders3 (x,y), gliders3 (x+40,y),
                         gliders3 (x,y+40), gliders3 (x+40,y+40)]

gliders5 :: Pos -> Scene
gliders5 (x,y) = scenes [gliders4 (x,y), gliders4 (x+80,y),
                         gliders4 (x,y+80), gliders4 (x+80,y+80)]

gliderGunL :: Pos -> Scene
gliderGunL (x,y) = [ (x+2,y+6), (x+2,y+7), (x+3,y+6), (x+3,y+7),
                     (x+12,y+6), (x+12,y+7), (x+12,y+8), (x+13,y+5),
                     (x+13,y+9), (x+14,y+4), (x+14,y+10), (x+15,y+4),
                     (x+15,y+10), (x+16,y+7), (x+17,y+5), (x+17,y+9),
                     (x+18,y+6), (x+18,y+7), (x+18,y+8), (x+19,y+7),
                     (x+22,y+4), (x+22,y+5), (x+22,y+6), (x+23,y+4),
                     (x+23,y+5), (x+23,y+6), (x+24,y+3), (x+24,y+7),
                     (x+26,y+2), (x+26,y+3), (x+26,y+7), (x+26,y+8),
                     (x+36,y+4), (x+36,y+5), (x+37,y+4), (x+37,y+5)]
```

A.1.3 Life.Engine.Hutton

```
module Life.Engine.Hutton where

import Data.List (nub,(\\))

import Life.Types

type Board = LifeBoard Config [Pos]

neighbs :: Config -> Pos -> [Pos]
neighbs c@((w,h),warp) (x,y) = let neighbors = [(x-1,y-1), (x,y-1), (x+1,y-1),
                                                (x-1,y),          (x+1,y),
                                               (x-1,y+1), (x,y+1), (x+1,y+1)]
    in if warp
        then map (\(x,y) -> (x `mod` w, y `mod` h)) neighbors
        else filter (\(x,y) -> (x >= 0 && x < w) && (y >= 0 && y < h)) neighbors

isAlive :: Board -> Pos -> Bool
isAlive b p = elem p $ board b

isEmpty :: Board -> Pos -> Bool
isEmpty b = not . (isAlive b)

liveneighbs :: Board -> Pos -> Int
liveneighbs b = length . filter (isAlive b) . (neighbs (config b))

survivors :: Board -> Board
survivors b =
    LifeBoard (config b) $ filter (\p -> elem (liveneighbs b p) [2,3]) $ board b

births :: Board -> Board
births b =
    LifeBoard (config b) $ filter (\p -> isEmpty b p && liveneighbs b p == 3)
                                $ nub $ concatMap (neighbs (config b)) $ board b

instance Life Board where
    empty c = LifeBoard c []
    alive = board
    inv p b = LifeBoard (config b) $ if isAlive b p
                                     then filter ((/=) p) $ board b
                                     else p : board b
    next b = LifeBoard (config b) $ board (survivors b) ++ board (births b)
```

A.1.4 Life.Display.Console

```
module Life.Display.Console where

import System.Console.ANSI
import Control.Concurrent

import Life.Types

cls :: IO ()
cls = putStr "\ESC[2J"

goto :: Pos -> IO ()
goto (x,y) =  putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")

writeat :: Pos -> String -> IO ()
writeat p xs = do
        goto p
        putStr xs

showcells :: Life board => board -> IO ()
showcells b = sequence_ [ writeat p "O" | p <- alive b ]

-- Runs indefinitely
lifeConsole :: Life board => board -> IO ()
lifeConsole b = do
        cls
        showcells b
        threadDelay (50 * 1000)
        lifeConsole (next b)

-- Runs for a given number of generations
lifeXConsole :: Life board => Int -> board -> IO ()
lifeXConsole 0 b = do
        cls
        showcells b
lifeXConsole n b = do
        cls
        showcells b
        lifeXConsole (n-1) $ next b
```

A.1.5 Game of Life Program

```
module Main where

import Life.Types
import Life.Engine.Hutton
import Life.Display.Console
import Life.Formations

-- Runs Life indefinitely
life :: Config -> Scene -> IO ()
life c s = lifeConsole (scene c s :: Board)

-- Runs Life for the specified number of generations
lifeX :: Int -> Config -> Scene -> IO ()
lifeX x c s = lifeXConsole x (scene c s :: Board)

originalLife = life ((20,20),True) $ glider (0,0)

main = life ((160,160),True) $ gliders5 (0,0)
```

A.2 Life Transformation Modules

The transformation modules featured here are:

1. `HERMIT.Set.Life` Defines the list to set transformation module

2. `HERMIT.QTree.Life` Defines the list quad-tree transformation module

A.2.1 Set Transformation Module

```
module HERMIT.Set.Life where

-- Libraries required for Hermit transformations
import Life.Types
import Data.Set as Set
import Data.List as List (nub,(\\))

type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Set Pos)
```

```
{-# NOINLINE repb #-}
repb :: [Pos] -> Set Pos
repb = fromList

{-# NOINLINE absb #-}
absb :: Set Pos -> [Pos]
absb = toList

{-# NOINLINE repB #-}
repB :: Board -> Board'
repB b = LifeBoard (config b) $ repb (board b)

{-# NOINLINE absB #-}
absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb (board b)

-- empty transformers
repcB :: (Config -> Board) -> Config -> Board'
repcB f c = repB (f c)

abscB :: (Config -> Board') -> Config -> Board
abscB f c = absB (f c)

-- alive transformers
repBs :: (Board -> Scene) -> Board' -> Scene
repBs f b = f (absB b)

absBs :: (Board' -> Scene) -> Board -> Scene
absBs f b = f (repB b)

-- isAlive and isEmpty transformers
repBpb :: (Board -> Pos -> Bool) -> Board' -> Pos -> Bool
repBpb f b p = f (absB b) p

absBpb :: (Board' -> Pos -> Bool) -> Board -> Pos -> Bool
absBpb f b p = f (repB b) p

-- liveneighbs transformers
repBpi :: (Board -> Pos -> Int) -> Board' -> Pos -> Int
repBpi f b p = f (absB b) p

absBpi :: (Board' -> Pos -> Int) -> Board -> Pos -> Int
absBpi f b p = f (repB b) p

-- inv transformers
reppBB :: (Pos -> Board -> Board) -> Pos -> Board' -> Board'
reppBB f x b = repB (f x (absB b))
```

```
abspBB :: (Pos -> Board' -> Board') -> Pos -> Board -> Board
abspBB f x b = absB (f x (repB b))

-- survivors, births, and next transformers
repBB :: (Board -> Board) -> Board' -> Board'
repBB f b = repB (f (absB b))

absBB :: (Board' -> Board') -> Board -> Board
absBB f b = absB (f (repB b))


-- GHC Rules for HERMIT
-- Simplification rules
{-# RULES
"repb/absb-fusion" [~] forall b. repb (absb b) = b
"absb/repb-fusion" [~] forall b. absb (repb b) = b
"LifeBoard-reduce" [~] forall b. LifeBoard (config b) (board b) = b
 #-}

--Code replacement rules
{-# RULES
"empty-l/empty-s" [~]
    repb [] =
    Set.empty
"not-elem/notMember" [~] forall p b.
    not (elem p (absb b)) =
    notMember p b
"elem/member" [~] forall p b.
    elem p (absb b) =
    member p b
"cons/insert" [~] forall p b.
    p : (absb b) =
    absb (insert p b)
"filter/delete" [~] forall p b.
    Prelude.filter ((/=) p) (absb b) =
    absb (delete p b)
"filter-l/filter-s" [~] forall f b.
    Prelude.filter f (absb b) =
    absb (Set.filter f b)
"nub-concatMap/unions" [~] forall f b.
    nub (concatMap f (absb b)) =
    absb (unions (toList (Set.map (fromList . f) b)))
"concat/union" [~] forall b1 b2.
    absb b1 ++ absb b2 =
    absb (union b1 b2)
 #-}
```

A.2.2 Quad-Tree Transformation Module

```
module HERMIT.QTree.Life where

import Life.Types
import Data.QuadTree
import Data.Boolean (xor)
import Data.List ((\\),sort,nub)

type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (QuadTree Bool)

{-# NOINLINE indices #-}
indices :: Size -> [Pos]
indices (w,h) = [ (x,y) | x<-[0..w-1], y<-[0..h-1] ]

{-# NOINLINE repb #-}
repb :: Size -> [Pos] -> QuadTree Bool
repb sz = foldr (\p qt -> setLocation p qt True) $ makeTree sz False

{-# NOINLINE absb #-}
absb :: QuadTree Bool -> [Pos]
absb qt = [ p | p <- indices (treeDimensions qt), getLocation p qt ]

{-# NOINLINE repB #-}
repB :: Board -> Board'
repB b = LifeBoard c $ repb (fst c) $ board b
    where c = config b

{-# NOINLINE absB #-}
absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb $ board b

repxB f = repB . f
absxB f = absB . f

repBx f = f . absB
absBx f = f . repB

repxBB :: (a -> Board -> Board) -> a -> Board' -> Board'
repxBB f x = repB . (f x) . absB

absxBB :: (a -> Board' -> Board') -> a -> Board -> Board
absxBB f x = absB . (f x) . repB

repBB :: (Board -> Board) -> (Board' -> Board')
repBB f = repB . f . absB
```

```
absBB :: (Board' -> Board') -> (Board -> Board)
absBB f = absB . f . repB


-- GHC Rules for HERMIT
-- Simplification rules
{-# RULES
"repb/absb-fusion" [~] forall s b.
    repb s (absb b) = b
"LifeBoard-reduce" [~] forall b.
    LifeBoard (config b) (board b) = b
 #-}

--Code replacement rules
{-# RULES
"empty-l/empty-t" [~] forall s.
    repb s [] = makeTree s False
"elem/getLocation" [~] forall p b.
    elem p (absb b) = getLocation p b
"cons/setLocation" [~] forall p b.
    p : (absb b) = absb (setLocation p b True)
"filter/setLocation" [~] forall p b.
    filter ((/=) p) (absb b) = absb (setLocation p b False)
"if-replace" [~] forall v p b.
    absb (setLocation p b (if v then False else True)) =
    absb (setLocation p b (not (getLocation p b)))
"filter/foldr-s" [~] forall f b.
    filter f (absb b) =
    absb (foldr (\p qt -> setLocation p qt (getLocation p b && f p))
                (makeTree (treeDimensions b) False)
                (indices (treeDimensions b)))
"filter/foldr-b" [~] forall f1 f2 b.
    filter f1 (nub (concatMap f2 (absb b))) =
    absb (foldr (\p qt -> setLocation p qt (f1 p))
                (makeTree (treeDimensions b) False)
                (indices (treeDimensions b)))
"concat/foldr" [~] forall f1 f2 b s.
    LifeBoard (config b)
        (repb s (absb (board (f1 b)) ++ (absb (board (f2 b))))) =
    LifeBoard (config b)
        (repb s (absb (foldr (\p qt -> setLocation p qt (getLocation p (board (f1 b))
                                                          || getLocation p (board (f2 b))))
                             (makeTree s False)
                             (indices s))))
 #-}
```